# Distance Vector Routing simulator

Francesco Bittasi: 0001071026

## Obiettivi

Il progetto mira a creare un'applicazione python che in semplicità simuli il protocollo di **Distance Vector Routing** per l'elaborazione delle tabelle di routing di vari nodi della rete, con l'obiettivo di *minimizzare il costo delle rotte* individuate.
L'obiettivo è di poter visualizzare gli aggiornamenti delle tabelle e l'invio dei dati passo per passo per comprendere meglio il funzionamento di base di questo protocollo.
La mia implementazione includerà le seguenti ottimizzazioni:

- **Split Horizon** -> ogni nodo pone il peso di una route ad *infinito* se questa passa attraverso il nodo a cui stiamo trasmettendo il *distance vector*.
- **Triggered Update** -> un nodo trasmette il proprio *distance vector* ai suoi vicini nel caso di aggiornamento della propria *routing table*.

## Implementazione

Il codice contenuto in `distanceVectorRouting.py` contiene 2 classi e un programma di esempio per mostrare il funzionamento del protocollo, se lanciato da terminale.

La classe `Network` tiene traccia della rete nel suo insieme, permette di aggiungere nodi e connessioni e gestisce la comunicazione dei dati tra un nodo e l'altro.

La classe `Node` si occupa invece di gestire un singolo nodo della rete, di ricevere ed aggiornare le sue rotte e trasmettere il proprio Distance Vector ai vicini.

### class Network

```
class Network
```

La classe Network rappresenta la rete in sé, tiene traccia degli oggetti della classe Nodi che la compongono e permette lo scambio di dati tra essi e il loro aggiornamenti. Tramite questa classe interagiamo col sistema creando nodi e connessioni.

```
def add_node(self, addr: str)
```

Aggiunge un nuovo nodo nella rete

```
def add_edge(self, node1: str, node2: str, weight: int)
```

Crea un nuovo collegamento tra due nodi informandoli della nuova connessione e invocandone lo scambio di distance vector.

Questo metodo può anche essere usato per aggiornare il peso di una route (se più efficiente).

```
def transmit(self, src: str, dst: str, dv: dict[str, tuple[int, str]])
```

Questo metodo viene invocato da un nodo ( `src` ) che intende inviare il proprio distance vector ( `dv` ) ad un nodo adiacente ( `dst` ).
Viene gestito dalla rete e non dai nodi stessi in quanto le classi dei nodi non hanno riferimenti agli oggetti, ma solo ai loro indirizzi.

## class Node

```
class Node
```

La classe Node rappresenta un singolo nodo della rete che tiene la propria routing table e l'aggiorna alla creazione/aggiornamento di un collegamento o al ricevimento di un distance vector.

```
def neighbour(self, node: str, weight)
```

Questo metodo aggiorna o crea una connessione con un nodo `node` vicino.
Se la connessione esiste già si occupa di aggiornare anche i pesi che hanno quel nodo come next hop.

```
def receiveDV(self, hop, dv: dict[str, tuple[int, str]])
```

Ricevuto un distance vector aggiorna la propria tabella di routing scegliendo i percorsi più efficienti.

```
def sendDV(self)
```

Invia il proprio distance vector a tutti i nodi vicini.

### ottimizzazioni

```
def updatedRT(self)
```

Questa funzione rappresenta il *Triggered Update*: se la tabella viene aggiornata viene chiamata questa funzione che lo notifica e richiama l'invio dei distance vector ai vicini.

```
def splitHorizon(self, addr: str)
```

Questa funzione viene chiamata all'invio di un distance vector, modifica il DV ponendo ad infinito tutte le routes che passano per il nodo a cui stiamo mandando i dati.

# Uso ed esempi

Se lanciato da terminale il programma esegue una simulazione generando 4 nodi e altrettante connessioni tra di loro.
Nell'esecuzione possiamo osservare tutti gli aggiornamenti che si verificano, le route generate, quelle scoperte, l'invio dei distance vector e l'aggiornamento delle tabelle.
Terminata questa fase vengono visualizzate a video lo stato di tutte le routing tables della rete, per poi andare ad aggiornare il peso di una delle connessioni e visualizzare gli effetti che questo ha sulla rete intera.

**Nodi e connessioni create:**

```
net.add_node("R1")
net.add_node("R2")
net.add_node("R3")
net.add_node("R4")

net.add_edge("R1", "R2", 8)
net.add_edge("R2", "R3", 4)
net.add_edge("R3", "R4", 21)
net.add_edge("R1", "R4", 3)
```

**Routing tables:**

```
NETWORK - routing tables
NODE:  R1  - routing table
addr  | weight | next_hop
R1  ->  0  via  R1
R2  ->  8  via  R2
R3  ->  12  via  R2
R4  ->  3  via  R4

NODE:  R2  - routing table
addr  | weight | next_hop
R2  ->  0  via  R2
R1  ->  8  via  R1
R3  ->  4  via  R3
R4  ->  11  via  R1

NODE:  R3  - routing table
addr  | weight | next_hop
R3  ->  0  via  R3
R2  ->  4  via  R2
R1  ->  12  via  R2
R4  ->  15  via  R2

NODE:  R4  - routing table
addr  | weight | next_hop
R4  ->  0  via  R4
R3  ->  15  via  R1
R2  ->  11  via  R1
R1  ->  3  via  R1
```

**Aggiornamento di una connessione:**

```
net.add_edge("R3", "R4", 1)
```

**Routing tables aggiornate:**

```
NETWORK - routing tables
NODE:  R1  - routing table
addr  | weight | next_hop
R1  ->  0  via  R1
R2  ->  8  via  R2
R3  ->  4  via  R4
R4  ->  3  via  R4

NODE:  R2  - routing table
addr  | weight | next_hop
R2  ->  0  via  R2
R1  ->  8  via  R1
R3  ->  4  via  R3
R4  ->  5  via  R3

NODE:  R3  - routing table
addr  | weight | next_hop
R3  ->  0  via  R3
R2  ->  4  via  R2
R1  ->  4  via  R4
R4  ->  1  via  R4

NODE:  R4  - routing table
addr  | weight | next_hop
R4  ->  0  via  R4
R3  ->  1  via  R3
R2  ->  5  via  R3
R1  ->  3  via  R1
```

# Difficoltà e considerazioni finali

Lo sviluppo è stato rallentato principalmente da due elementi: l'**algoritmo di aggiornamento** della *routing table* e la **sincronizzazione** degli invii dei *distance vectors*.

Nello sviluppo dell'algoritmo è stato necessario elaborare la nuova distanza delle route relative al nodo attuale che ha ricevuto il DV, e distinguere tra nuove rotte e route conosciute che possono essere aggiornate.
Inizialmente non era stato implementato il calcolo della distanza, perciò si riscontravano problemi in quanto venivano generate route inesistenti di peso 0 (le route di loopback venivano interpretate come route valide).

Questo progetto implementa l'ottimizzazione di *Triggered Update*, questo comporta, all'aggiornamento della tabella, l'invio del proprio distance vector a tutti i nodi vicini. Questo è risultato un problema in fase di creazione di un nuovo collegamento tra due nodi in quanto un nodo cercava di comunicare al nuovo vicino il proprio distance vector, quando questo ancora non era stato informato della nascita della nuova connessione.
Per risolvere questo problema si è deciso di affidare l'evento di invio dei distance vector, in seguito alla creazione di una connessione, alla network e di non lasciarlo in autonomia ai nodi. In questo modo l'invio dei dati era assicurato avvenire solamente dopo che entrambi i nodi avevano preso conoscenza della nuova connessione.

In questa implementazione del simulatore manca la gestione di eliminazione delle connessioni o il loro incremento di peso, in quanto richiederebbe una leggera ristrutturazione della classe dei nodi: questi dovrebbero tenere traccia separatamente dei collegamenti diretti stabiliti.
Attualmente infatti collegamenti diretti e indiretti sono salvati indifferentemente nella routing table, e vengono distinti solamente dal fatto che il 'next hop' corrisponde al destinatario nella rotta.
Questo comporta che, nel caso in cui una route indiretta diventi più conveniente di quella diretta, si va a perdere traccia del collegamento diretto stabilito in quanto questo viene sostituito (è stato deciso di non implementarlo in quanto si può assumere che, in linea di massima, route dirette sono sempre più efficienti di route indirette).
Questa sostituzione è problematica nel caso di un aumento di costo delle connessioni, e di conseguenza anche nell'eliminazione di queste (per farla si pone distanza = infinito), perché sarebbe impossibile ricostruire route alternative più efficienti a causa della perdita di informazioni sulla topologia.
In un'implementazione futura basterebbe avere una struttura che tiene traccia separatamente di tutte le connessioni dirette stabilite e le richiami in caso di bisogno, o che la routing table ammettesse più route per lo stesso destinatario (cosa ora impossibile in quanto stiamo usando un 'dizionario' di python).

Il progetto in conclusione ci ha permesso di visualizzare con successo il funzionamento base del protocollo di routing di distance vector, osservando passo passo come le tabelle dei router vengono aggiornate grazie ai dati condivisi tra nodi adiacenti.